TP de traitement des formules propositionnelles

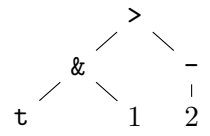
Ce TP introduit les concepts du cours de traitements des langages informatiques dans le cas particulier du langage des formules propositionnelles. Ce TP sert donc aussi d'introduction au cours de logique qui sert ici d'application du cours de TL. [1]

Les formules propositionnelles sont des *expressions* qui ne contiennent que des opérations et des variables propositionnelles. Concrètement, une telle formule est ici constituée à partir
- des constantes `'t'` (vrai) et `'f'` (faux) ;
- des variables booléennes, nommées en utilisant des entiers strictement positifs ;
- de la négation logique `'-'` (ainsi "`-t`" vaut "`f`") ;
- et des opérateurs binaires suivants : `'&'` (et logique), `'|'` (ou logique), `'>'` (implication logique).

Par exemple, "`(t & 1) > -2`" se lit *"vrai et 1 impliquent non 2"*. Ce langage est donc un des plus simples qu'on puisse imaginer. Ceci dit, il est suffisamment expressif pour exprimer des problèmes comme celui du *sudoku* qui va nous servir de cas de test significatif (cf. section 5).

On appelle *analyseur syntaxique* (ou *parser* en anglais) un programme informatique capable de lire une suite de caractères comme "`(t & 1) > -2`" et de l'interpréter comme une formule logique. Écrire un tel programme est un problème *à priori* non trivial, mais qui a des solutions bien comprises depuis la fin des années 1960. L'objectif du cours de période 2 est de présenter une de ces solutions (parmi les plus simples). Mais, avant d'en arriver là, on va commencer par se placer *en aval* de l'analyse syntaxique, c'est-à-dire en supposant que celle-ci a déjà été réalisée !

On commence donc par étudier le résultat de l'analyseur syntaxique lorsque la suite de caractères en entrée correspond bien à une formule syntaxiquement correcte. Ce résultat est un arbre appelé *arbre de syntaxe abstraite* (dessiné ci-contre) qui représente la structure du *parenthèsage* de la formule.



Ainsi, les différents concepts de TL qu'on va illustrer au travers de ce TP programmé en Java sont, par ordre d'apparition :
- les arbres de syntaxe abstraite ou *AST* (pour *Abstract Syntax Trees*) en sections 1 et 3 ;
- l'évaluation d'expression (ou interprétation d'expression) en section 2 ;
- la compilation, au sens d'une traduction d'une structure d'AST en une autre structure d'AST en sections 3 et 4 ;
- l'analyse syntaxique en notation préfixe en section 6 ;
- et l'analyse syntaxique en notation infixe via une grammaire hors-contexte LL(1) en section 7.

La prise en main des fichiers Java fournis est détaillée dans l'annexe section 8.

1 La notion d'AST illustrée sur les formules propositionnelles

On introduit une notation mathématique pour définir les AST indépendamment de la façon de les représenter dans tel ou tel langage de programmation. Cette notation sera formellement définie en cours. On se contente ici de la présenter de manière intuitive.

1. Historiquement, beaucoup des concepts de TL sont d'ailleurs issus de la formalisation de la logique.

1.1 Une notation pour les structures d'AST

Dans le cours, on formalise une structure d'AST comme une *signature multisortée représentable* dont le rôle est de définir *la forme possible des arbres*. Une telle signature est donnée par un ensemble fini de *sortes* (qui nomment des types d'arbre), un ensemble fini de *constructeurs* (qui correspondent à des noms écrits sur les nœuds) et un type (ou profil) pour chaque constructeur. Ce type permet de considérer le constructeur comme une fonction qui, étant donné un n-uplet d'arbres (avec éventuellement $n = 0$), retourne un nouvel arbre dont ceux-là sont les fils. Concrètement, ce type donne la sorte attendue de chaque fils du constructeur, ainsi que la sorte des arbres dont le constructeur est racine.

Dans le cas des formules propositionnelles, on utilise deux sortes : **Prop** pour le type des formules et **Pos** pour le type des entiers strictement positifs (qui représentent des noms de variables). On introduit ensuite un constructeur pour chaque opération ou constante du langage aisément identifiable à partir du nom du constructeur (e.g. $True$ pour `'t'` et And pour `'&'`).

$$
\begin{aligned}
True &: \ \textbf{Prop} \\
False &: \ \textbf{Prop} \\
Neg &: \ \textbf{Prop} \to \textbf{Prop} \\
And &: \ \textbf{Prop} \times \textbf{Prop} \to \textbf{Prop} \\
Or &: \ \textbf{Prop} \times \textbf{Prop} \to \textbf{Prop} \\
Implies &: \ \textbf{Prop} \times \textbf{Prop} \to \textbf{Prop} \\
Var &: \ \textbf{Pos} \to \textbf{Prop}
\end{aligned}
$$

Le cas le moins évident est celui de Var utilisé pour représenter les variables. Ici, **Pos** est qualifiée de *sorte externe* : la signature ne dit pas comment fabriquer des éléments de ce type (on suppose qu'on sait le faire par ailleurs). Tandis que **Prop** est une *sorte interne*. On ne peut fabriquer des arbres de type **Prop** qu'en utilisant les constructeurs de la signature.

Avec cette notation, l'arbre associé à l'exemple introductif "`(t & 1) > -2`" s'écrit [2]

$$Implies(And(True, Var(1)), Neg(Var(2)))$$

Une façon de coder cette signature en Java est donnée par le squelette de classe ci-dessous. La casse a été ici adaptée de manière à respecter les conventions de programmation Java.

La sorte **Prop** est représentée par une classe `Prop`, tandis que **Pos** est représentée abusivement par **int**. Les constructeurs sans arguments sont codés comme des constantes Java. Le constructeur Var qui n'a pas d'argument de sorte **Prop** est représenté par une méthode statique. Les autres constructeurs sont représentés comme des méthodes virtuelles dont le premier argument est implicitement l'objet auquel elle s'applique.

```
public abstract class Prop {
  final static public Prop TRUE;
  final static public Prop FALSE;
  public Prop neg();
  public Prop and(Prop);
  public Prop or(Prop);
  public Prop implies(Prop);
  static public Prop var(int);
}
```

Ainsi la formule de l'exemple introductif est codée comme un objet de type `Prop` construit par

```
Prop.TRUE.and(Prop.var(1)).implies(Prop.var(2).neg())
```

On aurait évidemment pu choisir un codage différent, en n'utilisant par exemple que des méthodes statiques et pas de méthode virtuelle.

Attention ici, à ne pas confondre le mot "*constructeur*" d'une signature, employé dans le contexte des signatures multisortées, et la notion de constructeur Java. Bien que les deux notions soient reliées (voir section 1.3), il ne faut pas les identifier.

Le principal intérêt d'une telle structure d'AST est de permettre la programmation des traitements

2. En cours, on l'écrira plutôt sans parenthèse "$Implies\ And\ True\ Var\ 1\ Neg\ Var\ 2$". En effet, avec les constructeurs écrits *avant* leurs arguments, il n'y a qu'une façon de parenthéser qui respecte le typage des constructeurs.

sur le langage indépendamment de l'analyse syntaxique : on détaille une façon de programmer ces traitements en section 1.3.

Le deuxième intérêt de cette structure est qu'elle facilite l'écriture d'un programme Java qui génère une formule de type `Prop`. En effet, le *compilateur* Java vérifie *statiquement* que les formules générées sont bien formées pour toute exécution du programme Java. Par exemple, on ne peut pas écrire de programme Java compilable qui génèrerait une formule mal formée correspondant à "`- & 1`". L'analyseur syntaxique va être lui-même un exemple d'un tel programme.

1.2 Exemple du sudoku

Un autre exemple de programme qui génère une formule `Prop` est donné par la méthode `initSudokuRule` du fichier fourni `Sudoku.java`. Étant donné un entier n (valant typiquement 4 ou 9), cette méthode calcule une formule qui exprime la règle du jeu du sudoku à n^2 cases (disponible en français sur http://www.e-sudoku.fr/regle-grille-sudoku.php).

Ici, l'état d'une grille de sudoku est codé par n^3 variables propositionnelles : à chacune des n^2 cases de la grille, on associe n variables telles que pour $p \in 1..n$, la p-ième variable de la case vaut "vrai" ssi l'entier p est écrit sur la case. C'est la méthode `var` dans ce même fichier qui exprime ce codage des noms de variable.

```
/** Nom de variable associé à un choix de nombre sur une case
 * @param i numéro de ligne (requiert 0 <= i < n)
 * @param j numéro de colonne (requiert 0 <= j < n)
 * @param p nombre écrit sur la case (requiert 1 <= p <= n)
 * @return nom de variable correspondant au choix de poser p en case (i,j)
 */
public static Prop var(int p, int i, int j) {
    return Prop.var((i*n+j)*n+p) ;
}
```

La formule retournée par `initSudokuRule` exprime la règle que sur la grille, un nombre p donné de $1..n$ apparaît au plus une fois par ligne, au plus une fois par colonne et au plus une fois par sous-région $\sqrt{n} \times \sqrt{n}$. Cette formule contient donc $\Theta(n^4)$ connecteurs logiques (il serait fastidieux de l'écrire à la main).

En section 2, on utilise le résultat de `initSudokuRule` pour vérifier qu'une grille de sudoku stockée dans un fichier respecte ces contraintes.

En section 5, on s'intéresse à la résolution des puzzles de sudoku : on utilise pour cela la formule retournée par `initSudokuPuzzle` qui exprime à quelle condition une grille est une solution d'un puzzle. Cette formule est simplement obtenue comme le et-logique de la formule retournée par `initSudokuRule` et de la formule exprimant que toute case de la grille contient au moins un nombre p de $1..n$.

1.3 Patron de conception interpréteur en Java

La classe `Prop` est implémentée par l'intermédiaire de sous-classes dont la hiérarchie suit la signature multisortée ci-dessus. Ce style de programmation OO s'appelle "*patron de conception interpréteur*".[3]

Chaque *sorte interne* (ici, il n'y a que `Prop`) correspond à une *classe abstraite*. Et, chaque *constructeur* de la signature ayant un type "$S_1 \times \ldots \times S_n \to S_{n+1}$" correspond à une *classe concrète*

— qui hérite de la classe correspondant à S_{n+1} ;
— a une liste d'attributs de type $S_1 \ldots S_n$;

3. cf http://en.wikipedia.org/wiki/Interpreter_pattern

— dont le *constructeur* (Java) a une liste d'arguments correspondant à $S_1 \times \ldots \times S_n$ qui permet d'initialiser la liste d'attributs.

Voir le code esquissé ci-dessous correspondant respectivement aux constructeurs *And* et *Var*.

```
class And extends Prop {
    final Prop left, right;
    And(Prop left, Prop right) {
        this.left=left ;
        this.right=right ;
    }
    ...
}
```

```
class Var extends Prop {
    final int value;
    Var(int value) {
        assert value>0 ;
        this.value=value ;
    }
    ...
}
```

Dans le fichier fourni `Prop.java`, les sous-classes de `Prop` sont privées (et statiques) : alternativement, on aurait pu utiliser un mécanisme de paquetage pour réaliser cette encapsulation. Au niveau de la classe `Prop`, les méthodes `and` et `var` sont simplement des appels à ces constructeurs Java.

```
public Prop and(Prop right) {
    return new And(this, right) ;
}
```

```
static public Prop var(int value) {
    return new Var(value) ;
}
```

Dans chacune de ces sous-classes, les méthodes virtuelles déclarées au niveau de `Prop` peuvent ainsi être implémentées en fonction du nœud représenté.

Typiquement, on illustre ce principe sur l'exemple de la méthode "**abstract void** `printPrefix();`" déclarée au niveau de la classe `Prop`, qui permet d'afficher l'AST sous forme textuelle en *notation préfixe*.[4] Le principe de cet affichage est le suivant : en chaque nœud, on affiche d'abord l'opérateur correspondant au constructeur (exemple *True* pour `'t'` et *And* pour `'&'`), puis les fils récursivement (donc, avec un parcours en profondeur). On fait un cas particulier pour le constructeur *Var* pour lequel on n'affiche que le fils. Dans le cas de l'exemple introductif, on obtient ainsi "`> & t 1 - 2`"[5]. Voir ci-dessous le code de cette méthode dans les sous-classes `And` et `Var`.

```
class And extends Prop {
   ...
   void printPrefix() {
       System.out.print("␣&") ;
       left.printPrefix() ;
       right.printPrefix() ;
   }
```

```
class Var extends Prop {
   ...
   void printPrefix() {
       System.out.print("␣"
                        + value);
   }
```

2 Tâche 1 : évaluation des formules propositionnelles

Cette tâche consiste à programmer un *évaluateur* (ou *interpréteur*) des formules propositionnelles. Concrètement, il s'agit d'étendre la classe `Prop` avec une méthode `eval` en respectant le principe du patron interpréteur décrit en section 1.3.

```
abstract public boolean eval(Environment env) ;
```

4. Ici, on simplifie un peu le code vis-à-vis du fichier fourni. Dans celui-ci `printPrefix` réalise un affichage indenté plus lisible, ce qu'on ne fait pas ici.

5. C'est donc très similaire à la notation sans parenthèse de l'AST.

Étant donné un environnement, c'est-à-dire une fonction associant une valeur booléenne à chaque nom de variables, *l'évaluation* d'une formule X est le booléen obtenu lorsqu'on remplace dans X chaque nom de variable par sa valeur dans l'environnement (et qu'on calcule l'expression booléenne sans variable ainsi obtenue). [6]

Sur l'exemple introductif "`(t & 1) > -2`", si l'environnement associe `t` aux variables 1 et 2, alors la formule s'évalue sur le booléen "`(t & t) > -t`" qui vaut "`t > f`" c'est-à-dire "`f`". Si l'environnement associe `t` à 1 et `f` à 2, alors la formule s'évalue sur "`(t & t) > -f`" qui vaut "`t > t`" c'est-à-dire "`t`".

Concrètement, l'environnement `env` en paramètre de `eval` est un objet de type `Environment` qui a une méthode `get` associant à chaque un booléen à chaque nom de variable :

```
public class Environment {
   public boolean get(int name) ; // requiert name > 0
}
```

2.1 Tests de base

Pour déboguer votre méthode `eval`, il faut la tester à partir des programmes fournis dans le fichier `Test.java`. La classe `Test` contient des attributs statiques définissant des cas des tests de type `TestCase`. Typiquement, l'exemple introductif du sujet est défini par :

```
public final static TestCase exintro = new TestExIntro();
```

avec

```
class TestExIntro extends TestCase {

        TestExIntro() { // initialise attribut "public Prop test"
                super(Prop.TRUE.and(Prop.var(1)).implies(Prop.var(2).neg()));
        }

        boolean oracle(Environment v) {
                return implies(true && v.get(1), !v.get(2));
        }
}
```

Ici la méthode `oracle(v)` simule le résultat attendu de `exintro.test.eval(v)`.

À partir de là, pour chaque traitement à tester et chaque cas de test, la classe `Test` définit un pilote au moyen d'une classe interne avec sa propre méthode `main`. Le nom du cas de test apparaît en suffixe dans le nom de la classe interne. Par exemple, pour tester l'affichage (préfixe) sur `exintro`, on lance [7] :

```
java -ea Test\$Printexintro
```

Pour tester la méthode `eval`, il faut utiliser les classes internes préfixées par `Eval`. Le `main` associé attends alors un liste de noms de variables (strictement positifs) en arguments de la ligne de commande. Il évalue alors le cas de test avec l'environnement qui associe `'t'` à une variable si et seulement si celle-ci figure dans la liste en argument.

6. La terminologie utilisée ici correspond s'emploie fréquemment pour les langages d'expression, comme les *expressions arithmétiques*. En logique, on utilise *interprétation* ou *modèle* à la place de *environnement*. Et l'évaluation correspond à la relation de *satisfaisabilité* : "*un modèle satisfait une formule*" revient donc au fait que l'évaluation de cette formule sur ce modèle répond *vrai*.

7. À lancer après avoir compilé et positionné le `CLASSPATH` comme expliqué en section 8.

Ainsi, la commande "`java -ea Test\$Evalexintro 1 2`" doit afficher "`eval = false`". Et, la commande "`java -ea Test\$Evalexintro 1`" doit afficher "`eval = true`". Si l'évaluation du cas de test diffère de sa méthode `oracle`, une erreur sera levée à l'exécution.

Vous devez tester ainsi votre méthode `eval` sur les différents cas de tests et sur différentes valeurs de l'environnement. Vous pouvez aussi ajouter d'autres cas de tests, notamment pour faciliter le débogage.

2.2 Tests des grilles de sudoku

Une fois que vous pensez que votre méthode `eval` fonctionne bien, vous pouvez l'utiliser sur un exemple plus significatif : la vérification qu'une grille de sudoku respecte les règles (données par `initSudokuRule`).

Pour commencer, on se limite aux grilles de sudoku de taille 4 (dans le fichier `Sudoku.java` fourni la constante n est fixée à 4) dans les fichiers "`sudoku4.*`". Syntaxiquement, ces fichiers sont construits en partant du fichier "`sudoku4.vide`" et en remplaçant certains caractères "`_`" (représentant une case vide de la grille) par un chiffre entre 1 et n. Une méthode statique de la classe `Sudoku` appelée `PuzzleReader.read` permet de construire l'environnement représentant une telle grille (`PuzzlePrinter.print` permet de réafficher un tel environnement sous forme de grille).

Pour tester une grille, il suffit d'exécuter la méthode `main` de la classe `Sudoku\$Checker` en passant la grille sur *l'entrée standard* :

```
java -ea Sudoku\$Checker < sudoku4.2169
```

Vérifiez que toutes les grilles sont valides, excepté celles en "`sudoku4.ko*`" où le numéro en suffixe est inférieur ou égal à 4. Les grilles `sudoku4.ko5` et `sudoku4.ko6` sont valides, mais sans solution.

3 Tâche 2 : syntaxe des formules en forme normale négative

En section 4, on implémente un algorithme élémentaire de simplification des formules propositionnelles : essentiellement, cette simplification consiste à éliminer les constantes `'t'` et `'f'` et l'opérateur `'>'`, et à propager les négations jusqu'aux feuilles de l'AST sous-jacent (les noms de variable).

Au préalable, dans cette section, on s'intéresse à une forme de formules simplifiées appelée *forme normale négative* (ou NNF acronyme de *Negative Normal Form*). Plus précisément, une formule **Prop** est en NNF si et seulement si

- elle n'a pas de sous-formule stricte syntaxiquement constante (mais la formule elle-même peut être réduite à une constante),
- et, elle ne contient pas l'opérateur binaire `'>'`,
- et, chaque opérateur `'-'` s'applique directement à un nom de variable.

Ainsi, l'exemple introductif n'est pas une NNF. Mais il a une NNF logiquement équivalente : "`-1 | -2`".

Dans la suite de cette section, on définit une sorte d'AST appelée **Nnf** qui représente très exactement la syntaxe des formules NNF. On emploie pour cela la même démarche que celle décrite en section 1. La tâche 2 consiste à implémenter la traduction des AST de sorte **Nnf** en AST de sorte **Prop** (ce qui permet donc d'appliquer ensuite les traitements implémentés au niveau de la classe `Prop`, comme `printPrefix` ou `eval`).

3.1 Définition des AST de sorte Nnf

Tout d'abord, on appelle *littéral*[8] une formule qui est soit réduite à un nom de variable, soit la négation d'un nom de variable. Ici, un tel littéral est directement codé par un entier non nul. On introduit donc la *sorte externe* **NNInt** comme ensemble des entiers non nuls.

On introduit aussi les *sortes internes* **Nonconstant** (pour désigner les NNF non constantes) et **Nnf** (pour désigner les NNF quelconques). On exprime ainsi directement les contraintes syntaxiques de la forme NNF par des règles de typage.

$$\begin{aligned} \mathit{True} &: \mathbf{Nnf} \\ \mathit{False} &: \mathbf{Nnf} \\ \mathit{Cast} &: \mathbf{Nonconstant} \to \mathbf{Nnf} \\ \mathit{And} &: \mathbf{Nonconstant} \times \mathbf{Nonconstant} \to \mathbf{Nonconstant} \\ \mathit{Or} &: \mathbf{Nonconstant} \times \mathbf{Nonconstant} \to \mathbf{Nonconstant} \\ \mathit{Literal} &: \mathbf{NNInt} \to \mathbf{Nonconstant} \end{aligned}$$

La signification des constructeurs se déduit directement de leur nom. En particulier, une formule $\mathit{Cast}(X)$ représente exactement la même formule que X.

L'implémentation de cet AST dans la classe `Nnf` suit la démarche décrite en section 1.3 à ceci près que le constructeur *Cast* n'est pas représenté comme une classe du fait de sa signification très particulière. En Java, on peut simplement le représenter en faisant de la classe abstraite `Nonconstant` une sous-classe de `Nnf`. Étant donné un objet `o` de type `Nonconstant`, "*Cast*(`o`)" est représenté par le cast Java "`(Nnf)o`" (on peut en fait laisser ce cast implicite en général).

3.2 À faire

La traduction des AST de sorte **Nnf** en sorte **Prop** est déclarée dans la classe `Nnf` par la méthode virtuelle :

```
abstract public Prop toProp() ;
```

Vous devez modifier le fichier `Nnf.java` fourni de manière à implémenter la méthode `toProp` qui retraduit une formule `Nnf` en formule `Prop`.

Testez votre implémentation sur le cas de test `mpx` du fichier `Test.java`.

4 Tâche 3 : mise-en-forme normale négative

La tâche 3 consiste à implémenter l'algorithme qui permet de traduire n'importe quelle formule de type `Prop` en une formule de type `Nnf` qui soit logiquement équivalente. Concrètement, cette traduction est effectuée dans la méthode suivante de la classe `Prop` :

```
public Nnf toNnf();
```

Ainsi, pour tout `x` de type `Prop` et tout `env` de type `Environment`, on veut

`x.eval(env)==x.toNnf().toProp().eval(env)`

Par ailleurs, pour `y` de type `Nnf`, l'arbre `y.toProp().toNnf()` doit être identique à l'arbre `y`.

8. traduit par "*literal*" en anglais

On attend aussi que le coût de votre algorithme `toNnf` soit linéaire en fonction du nombre de nœuds traversés (et même, on attend un seul parcours de l'arbre). De même, la taille de l'arbre produit (en nombre de nœuds) doit aussi être linéaire.

L'algorithme de mise-en-NNF est en fait séparé en deux parties. La première partie se concentre sur l'élimination des constantes. Celle-ci est faite par les méthodes virtuelles suivantes de la classe `Nnf` :

```
abstract public Nnf and(Nnf right) ; // équivalent à "this & right"
abstract public Nnf or(Nnf right) ;  // équivalent à "this | right"
```

Attention, contrairement à ce qui se passe dans la classe `Prop` ces méthodes ne correspondent pas directement aux constructeurs *And* et *Or* de la signature **Nnf**.

Dans la deuxième partie, on peut donc se concentrer sur la propagation de la négation vers les feuilles, en comptant sur les méthodes de la première partie pour éliminer les constantes. Concrètement, la deuxième partie consiste à implémenter la méthode virtuelle suivante de la classe `Prop` :

```
abstract public Nnf toNnf(boolean neg) ;
```

de telle façon que `x.toNnf()` puisse être implémentée par `x.toNnf(false)` et telle que `x.toNnf(true)` retourne une `Nnf` logiquement équivalente à `x.neg().toNnf()`.

Le rôle du booléen `neg` est de mémoriser si on doit propager une négation vers les feuilles ou pas, sachant qu'une double négation équivaut à aucune négation. De plus, lorsque la propagation d'une négation traverse le nœud un *And* ou un *Or*, on utilise les lois de De Morgan (voir http://en.wikipedia.org/wiki/De_Morgan's_laws) : un *And* se transforme en *Or* et réciproquement. Pour le nœud *Implies*, il suffit d'utiliser l'équivalence logique $Implies(X, Y)$ équivaut à $Or(Neg(X), Y)$ pour se ramener aux cas précédents.

Pour tester votre méthode `toNnf`, utilisez les classes internes de `Test` préfixées par `Nnf` sur les différents cas de test.

5 Tâche 4 : tests sur la résolution de grilles de sudoku

Lorque votre méthode de mise-en-NNF semble bien fonctionner, testez-là sur un *vrai* exemple : la résolution des grilles de sudoku. Pour cela, on va utiliser un *SAT-solver* appelé "`clasp`" [9]. Étant donné une formule propositionnelle dans une certaine forme (qu'on va préciser plus loin), ce logiciel répond soit `UNSATISFIABLE` (aucun environnement ne satisfait la formule), soit `SATISFIABLE` et dans ce cas affiche un environnement qui satisfait la formule. [10] Ce logiciel implémente une variante de l'algorithme DPLL [11] qui vous sera éventuellement présenté dans le cours de logique (voir avec l'enseignant concerné). La formule en entrée de cet algorithme doit être en CNF (pour "*Conjunctive Normal Form*" ou "*forme normale conjonctive*" en français). Une telle formule est une NNF dans laquelle toute sous-formule (ou sous-arbre) d'un nœud *Or* ne contient pas de nœud *And*.

La NNF de l'exemple introductif "`-1 | -2`" est bien une CNF. Par contre, une formule comme "`(1 & 2) | 3`" est une NNF qui n'est pas une CNF. Naïvement, pour mettre une formule NNF en CNF, il suffit d'appliquer la distributivité du *Or* sur le *And* : une formule "$Or(And(X, Y), Z)$" est traduite en la formule équivalente "$And(Or(X, Z), Or(Y, Z))$". Mais cet algorithme augmente la taille de la formule produite de manière exponentielle (cf. duplication du Z). Il existe d'autres algorithmes

9. Logiciel libre disponible sur http://www.cs.uni-potsdam.de/clasp/ ou comme paquet Ubuntu d'un dépôt `universe`.
10. Cet environnement est affiché sous la forme d'une liste de littéraux dans laquelle une variable négative a la valeur `'f'`, et une variable positive a la valeur `'t'` (un nom de variable ne pouvant apparaître qu'au plus une fois dans la liste).
11. http://en.wikipedia.org/wiki/DPLL_algorithm

qui produisent des formules de taille linéaire : ils ne sont pas très compliqués, mais hors-du-cadre de ce TP. Ici, on se contente donc de considérer des formules initiales dont la NNF est une CNF. Ce cadre est suffisant pour traiter les formules générées sur les sudokus.

Concrètement, `clasp` attends sur son entrée standard une formule CNF au format DIMACS [12]. On fournit ici une méthode `printDimacs` dans la classe `Nnf` qui effectue l'affichage de la formule dans ce format si celle-ci est une CNF et qui lève une exception sinon.

```
public void printDimacs() ;
```

La tâche 4 consiste donc à utiliser le script `sudoku.sh` (qui utilise les fichiers Java + `clasp`) pour résoudre des grilles de sudoku. Ce script commence par afficher la grille d'entrée. Il passe alors à `clasp` une formule exprimant la conjonction de la formule produite par `initSudokuPuzzle` (voir section 1.2) et d'une formule exprimant les contraintes de la grille d'entrée. S'il n'y a pas de solution, il s'arrête. Sinon, il affiche la grille correspondant à la solution trouvée par `clasp`. Puis il effectue un deuxième appel à `clasp` en lui passant une formule qui est la conjonction de la formule précédente et la négation de la solution trouvée par `clasp` : on peut ainsi vérifier qu'il n'y a qu'une solution pour la grille d'entrée conformément aux usages. Sinon, cette deuxième solution est affichée.

En utilisant la commande "`./sudoku.sh nom_de_grille`" (pour une taille de sudoku $n = 4$), effectuez les vérifications listées ci-dessous :

1. il y a (au moins) deux solutions pour la grille "`sudoku4.vide`" ;
2. il n'y a aucune solution sur la grille "`sudoku4.ko*`" ;
3. il y a une seule solution sur la grille "`sudoku4.2169`".

En cas de comportement différent, votre algorithme de mise-en-NNF est vraisemblablement faux. Il faut alors le déboguer avant de faire de nouveaux essais de résolution de sudoku : il est conseillé de créer pour cela de nouveaux (petits) cas de tests dans la classe `Test` plutôt que d'essayer de déboguer directement à l'aide des grosses formules de sudokus. Néanmoins, si vous souhaitez avoir une idée de la formule d'entrée vous pouvez lancer la commande ci-dessous (qui affiche en notation infixe la formule avant sa mise-en-NNF) :

```
java -ea Sudoku\$DebugSolver < nom_de_grille
```

Lorsque le script fonctionne pour les sudokus de taille 4, vous pouvez essayer ceux de taille 9 en positionnant dans le fichier `Sudoku.java` la constante `rn` à 3 (et en recompilant).

6 Tâche 5 : analyse syntaxique en notation préfixe

La section 1.3 décrit la méthode `printPrefix` qui affiche un AST de type `Prop` en notation préfixe. La tâche 5 consiste à programmer un *analyseur syntaxique* qui réalise en gros la réciproque : détecter si une suite de caractères sur l'entrée standard correspond à un AST de type `Prop` en notation préfixe, et dans ce cas retourner cet AST, sinon lever une erreur. La syntaxe reconnue par l'analyseur autorise des *commentaires* : ceux-ci commencent par le caractère `'%'` et se terminent en fin de ligne. Elle autorise un nombre arbitraires de *blancs* (caractère espace, ou tabulation, ou retour à la ligne) entre les *mots* consituants la formule. Dans la suite, on désigne ces mots sous le vocable de *lexèmes* (ou *tokens* en anglais). Un tel lexème est donc soit un nom de variable (constitué d'une suite de chiffres en base 10), soit un caractère parmi "`&|>-tf`".

Des exemples de fichiers en notation préfixe sont donnés dans le sous-répertoire `prefix/`. Les fichiers commençant par "`ok_`" doivent être acceptés par l'analyseur, alors que ceux commençant par "`ko_`"

12. http://www.satcompetition.org/2009/format-benchmarks2009.html

doivent être rejetés (ils contiennent des erreurs de syntaxe). Par exemple, le fichier `ok_exparser.prop` contient :

```
% exemple de formule en notation préfixe
& 421 -f
```

Il est constitué de 4 lexèmes : `&`, `421`, `-` et `f`. Et l'AST correspondant est à cette formule est : $And(Var(421), Neg(False))$.

Pour simplifier la programmation de l'analyseur syntaxique, celui-ci est décomposé en deux parties : l'analyseur lexical (ou *lexer* en anglais) qui transforme la suite de caractères en suite de lexèmes (cf. section 6.1) ; et l'analyse syntaxique proprement dite qui construit l'AST à partir de la suite de lexèmes (cf. section 6.2)

6.1 Analyse lexicale

On fournit ici l'analyseur lexical dans le fichier `Lexer.java`. Typiquement, la commande suivante permet d'afficher sur une ligne à part chaque lexème d'un fichier donné (NB le fichier est lu sur l'entrée standard) :

```
java -ea Lexer < prefix/ok_exparser.prop
```

En interne, pour chaque lexème *concret* (c'est-à-dire une certaine suite de caractères) reconnu, l'analyseur construit un lexème *abstrait* : c'est-à-dire, un objet Java du type fourni `Token`. Ainsi, l'analyseur lexical est en fait un objet de type `Lexer` qui fonctionne comme une sorte d'itérateur Java : il a essentiellement une méthode `next` qui déplace la tête de lecture jusqu'au prochain lexème et le retourne.

```
public Token next() ;
```

Ici, un lexème abstrait spécial joue le rôle de sentinelle de fin de fichier : il ne correspond à aucun lexème concret et sert uniquement à exprimer que la lecture du fichier d'entrée est finie. Un lexème abstrait de type `Token` (cf. profil ci-dessous) correspond donc

- soit la sentinelle de fin : dans ce cas, l'attribut `code` vaut la valeur `Token.EOF` ;
- soit à un nom de variable : dans ce cas, la méthode `isVar()` répond **true** et le nom de la variable est obtenu par la méthode `getVar()` ;
- soit un caractère ASCII : dans ce cas, la méthode `isVar()` répond **false** et l'attribut `code` vaut le code ASCII du caractère.

Voilà le profil de la classe `Token` :

```
public class Token {
    public boolean isVar() ;
    public int getVar() ;
    public final int code ;
    static final int EOF = -1 ;
}
```

Code exécuté par "`java -ea Lexer`" [13] :

```
Lexer lexer = new Lexer() ;
Token curr = lexer.next() ;
while (curr.code != Token.EOF) {
    System.out.println(curr) ;
    curr=lexer.next() ;
}
```

6.2 Principe de l'analyse syntaxique en notation préfixe

L'analyseur syntaxique invoque donc la méthode `next` du lexer pour construire l'AST au fur et à mesure de la lecture du fichier d'entrée : la suite des lexèmes n'est donc pas stockée en mémoire, et la

13. c'est-à-dire dans la méthode `main` de `Lexer`

lecture du fichier s'interrompt à la première erreur detectée dans le fichier.

Pour programmer l'analyseur syntaxique, on se base sur la propriété suivante de la notation préfixe :

> *Pour toute suite de lexèmes u, il existe* **au plus** *une suite de lexèmes v qui est un préfixe de u et qui correspond à l'écriture en notation préfixe d'un AST de sorte* **Prop**.

Par exemple, si u correspond à "`& t 1 - 2`", alors l'unique v possible correspond à "`& t 1`". Par contre, si u correspond à "`& t`", il n'y a aucun v possible.

Cette propriété permet en effet de programmer récursivement une méthode `recParse`, qui étant donné la suite u des lexèmes *restant à lire*,

- s'il existe un préfixe v de u tel que v est l'écriture en notation préfixe d'un AST de type `Prop`, alors `recParse` retourne cet AST et la tête de lecture du lexer après l'appel se trouve sur le premier lexème après la suite v dans u ;[14]
- si un tel v n'existe pas alors, `recParse` lève l'exception `Lexer.ErreurSyntaxe`.

```
Prop recParse () ;
```

L'unicité du v recherché permet en effet une programmation récursive simple de cette procédure : le premier lexème de v (et donc u) n'est pas la sentinelle de fin (sinon on lève l'exception) et correspond à un nœud dont on connaît le nombre de fils : 0 pour `'t'` ou `'f'` ou un nom de variable, 1 pour `'-'` et 2 pour les lexèmes parmis "`&|>`"). Pour chaque fils attendu, on effectue en séquence un appel récursif à `recParse` afin de récupérer l'AST correspondant. Si l'exception `Lexer.ErreurSyntaxe` est levée dans un des appels récursifs, c'est que le v n'existe pas : on laisse l'exception se propager. Dans le cas contraire, on retourne l'AST obtenu par assemblage du nœud initial et de ses fils.

La méthode principale de l'analyseur syntaxique se contente d'appeler `recParse` et de vérifier qu'en sortie de l'appel, on a bien atteint la fin du fichier.

```
Prop mainParse() ;
```

6.3 À faire

Votre travail consiste à compléter la méthode `recParse` (décrite ci-dessus) fournie dans le fichier `PrefixParser.java`. La méthode `Lexer.nextVar` fournie contient des bogues que vous devez corriger (après avoir compris ce qu'elle est censée faire). [15]

Pour tester que le parser fonctionne, on commencera par utiliser le jeu de tests du sous-répertoire `prefix/` sur l'exécutable "`java -ea PrefixParser -prefix`". Celui-ci commence par l'analyse syntaxique de *l'entrée standard*. Si celle-ci échoue alors un message d'erreur est affiché. Sinon, l'AST construit est réaffiché en syntaxe préfixe. Au cours des tests, vérifiez que :

- les fichiers préfixés par "`ko_`" provoque le message d'erreur indiqué en commentaire en début du fichier ;
- les fichiers préfixés par "`ok_`" sont réaffichés sans erreur.

Une fois que tous ces tests passent, essayez un test plus conséquent en lançant le script suivant (pour afficher le mode d'emploi, lancer ce script sans option)

```
./testparser_sudoku.sh -prefix
```

Celui-ci teste le parser sur la règle du jeu du Sudoku (pour afficher le mode d'emploi complet, lancer ce script sans option). Essayer d'abord avec `Sudoku.rn=2`. Si ça marche, essayez avec `Sudoku.rn=3` pour vérifier le passage à l'échelle de votre parser.

14. Autrement dit, la suite u' des lexèmes qui restent à lire après l'appel récursif vérifie $u = v\ u'$.
15. Comme cette méthode fonctionne à peu près, vous pouvez commencer par implémenter `recParse`, et ne corriger `Lexer` que dans un deuxième temps. Implémenter `recParse` vous aidera à comprendre ce que doit faire `Lexer`.

7 Tâche 6 : analyse syntaxique en notation infixe

Pour cet analyseur, on réutilise le lexer défini en section 6.1. Le fonctionnement de l'analyseur va donc fortement ressembler à celui décrit en section 6.2, mais pour une syntaxe concrète plus complexe.

Pour écrire ce parser, il faut commencer par effectuer une série de transformations de BNF à partir de la spécification initiale du parser donnée en 7.1. Ici, il faut s'appuyer sur les concepts vus en cours (qui ne sont pas rédétaillés ici).

Précisons qu'il existe des outils pour faire ces transformations automatiquement, de sorte que le développeur peut directement écrire son parser dans un format proche de celui de la spécification donnée en 7.1. Mais le but de ce TP est justement de faire ces transformations "à la main" sur un petit exemple afin d'acquérir la compréhension du fonctionnement de ces outils qui s'avère indispensable quand on les utilise sur des exemples plus complexes.

7.1 Spécification de la syntaxe infixe

La spécification du parser infixe est donnée par la BNF attribuée ci-dessous et les règles de précédence données plus loin. Cette BNF comporte un seul non-terminal (et axiome) noté **P**. Ses terminaux sont pos et tous les symboles entre apostrophes. Le système d'attribut spécifie les AST associés aux mots reconnus par le parser. Les profils des symboles de la BNF sont donnés par

$$\mathbf{P}\uparrow\mathbf{Prop} \qquad \text{pos}\uparrow\mathbf{Pos}$$

La BNF attribuée est

$\mathbf{P}\uparrow p$	::=	`'t'`	$p := True$
	|	`'f'`	$p := False$
	|	pos$\uparrow v$	$p := Var(v)$
	|	`'-'` $\mathbf{P}\uparrow p_1$	$p := Neg(p_1)$
	|	`'('` $\mathbf{P}\uparrow p$ `')'`	
	|	$\mathbf{P}\uparrow p_1$ `'&'` $\mathbf{P}\uparrow p_2$	$p := And(p_1, p_2)$
	|	$\mathbf{P}\uparrow p_1$ `'|'` $\mathbf{P}\uparrow p_2$	$p := Or(p_1, p_2)$
	|	$\mathbf{P}\uparrow p_1$ `'>'` $\mathbf{P}\uparrow p_2$	$p := Implies(p_1, p_2)$

Cette BNF ci-dessus est ambiguë : pour un même mot, plusieurs AST peuvent être synthétisés. Pour désambiguïser, on restreint les arbres d'analyses possibles en utilisant les règles de précédences ci-dessous :

— les opérateurs ont les niveaux de précédence suivants

opérateur	précédence
`'>'`	3
`'|'`	2
`'&'`	1
`'-'`	0

— comme dans tout langage d'expression, les constantes, les variables et les parenthèses sont au niveau de précédence 0

— tous les niveaux de précédences sont associatifs à droite

7.2 A faire : transformer la BNF en une BNF non-ambiguë

Votre premier travail consiste à écrire une BNF attribuée non-ambiguë, qui reconnaisse le même langage que celle donnée en section 7.1, et qui, pour chaque mot de ce langage, synthétise le même AST lorsqu'on restreint la BNF 7.1 aux arbres d'analyse qui satisfont les règles de précédence.

Pour exprimer les contraintes de précédence comme des règles de BNF, il faut introduire un non-terminal $\mathbf{P}_n$ par niveau de précédence n (avec $0 \leq n \leq 3$) où chaque non-terminal a la signification suivante :

> La langage reconnu par $\mathbf{P}_n$ est l'ensemble des mots u reconnus par le $\mathbf{P}$ (de la BNF en 7.1) tels que pour tout sous-mot v de u contenant un opérateur de niveau de précédence strictement supérieure à n, il existe un sur-mot v' de v tel que `'('` v' `')'` est un sous-mot de u qui est reconnu par $\mathbf{P}$.

Ainsi, $\mathbf{P}_3$ correspond à l'ensemble des mots reconnus par $\mathbf{P}$ (et donc est l'axiome de votre BNF). Et, pour $0 \leq n < 3$, on a $\mathbf{P}_n \subset \mathbf{P}_{n+1}$. De plus, si u_1 et u_2 sont dans $\mathbf{P}$, alors un mot de la forme "u_1 `'|'` u_2" peut être reconnu par $\mathbf{P}_2$ (à condition que u_1 et u_2 soient eux-mêmes dans $\mathbf{P}_2$) mais pas par $\mathbf{P}_1$.

On ne demande pas de prouver à ce niveau que la BNF obtenue est non-ambiguë. Cela va être vérifié de manière indirecte en la transformant en BNF LL(1).

7.3 A faire : transformer la BNF non-ambiguë en une BNF LL(1)

Vous devez maintenant transformer la BNF de la section 7.2 en BNF LL(1) : vous devez obtenir une BNF attribuée LL(1) qui reconnaît le même langage et synthétise les mêmes AST que celle de la section 7.2.

Comme il n'y a que des opérateurs associatifs à droite, il suffit en principe de factoriser les règles à gauche. En cas d'opérateurs associatifs à gauche, il faudrait éliminer les récursions immédiates à gauche.

Vérifiez que la BNF est bien LL(1) en effectuant le calcul de directeurs.

7.4 A faire : implémenter le parser LL(1)

Votre travail consiste maintenant à implémenter le parser spécifié section 7.1 en le dérivant de la BNF LL(1) obtenue en section 7.3. Concrètement, il s'agit de compléter le fichier fourni `InfixParser.java`, en modifiant la méthode `parseProp` de manière à ce qu'elle accepte tout préfixe de l'entrée qui est un sous-mot de $\mathbf{P}_3$ et produise l'AST associé. Comme vu en CTD, il faut éventuellement introduire une méthode spécifique pour chaque autre non-terminal de la BNF LL(1).

Pour tester que le parser fonctionne, commencez par utiliser le jeu de tests du sous-répertoire `infix/` à partir du script `testinfix.sh`. Une fois que tous ces tests fonctionnent correctement, essayez un test plus conséquent avec "`./testparser_sudoku.sh -infix`" d'abord avec `Sudoku.rn=2`, puis `Sudoku.rn=3`.

8 Annexe : prise en main des fichiers Java fournis

Avant de lancer une première exécution Java des sources fournies (situées dans le sous-répertoire `src/`), il faut :

1. Compiler en lançant `ant`. Son "makefile" est donné dans `build.xml`. Il place les fichiers "`.class`" produits dans le sous-répertoire `bin/`.
2. Positionner correctement la variable `CLASSPATH` en lançant simplement "`source prelude.sh`".

Exécuter la machine virtuelle java avec l'option "`-ea`" pour vérifier à l'exécution les assertions (instruction **assert**) présentes dans le code. Pour chercher le nom des classes éventuellement exécutables, pensez à faire "`ls bin/`".

8.1 Édition des fichiers Java

Vous pouvez éditer les fichiers Java avec votre éditeur favori directement dans le sous-répertoire `src/`. Pour compiler, il faut utiliser `ant` à la racine du répertoire fourni comme expliqué ci-dessus. Vous pouvez aussi éditer/compiler les fichiers sous `eclipse`. Il suffit de lancer "`eclipse &`" en ligne de commande, puis de sélectionner le menu

`File/New>Project.../Java Project`

puis de décocher l'option "`Use default location`", puis de sélectionner le répertoire fourni via le bouton "`Browse`". À priori, l'organisation du répertoire est alors compatible avec les options par défaut de `eclipse`.

8.2 Brève description des classes Java fournies (voir fichiers de `src/`)

- La classe `Erreur` déclare l'unique exception directement levée par les sources fournies (en dehors de `Lexer` et `PrefixParser`).
- La classe `Prop` (A COMPLETER sections 2 et 4) définit la structure d'AST des formules propositionnelles.
- La classe `Environment` définit les environnements et les entrées-sorties sur les environnements.
- La classe `Test` fournit quelques tests simples sur les manipulations d'AST.
- La classe `Nnf` (A COMPLETER sections 3 et 4) définit une structure d'AST pour représenter des formules sous forme NNF.
- La classe `Sudoku` fournit les tests à partir des grilles de Sudoku.
- La classe `PrefixParser` (A COMPLETER section 6) fournit le squelette d'analyseur syntaxique en notation préfixe.
- La classe `Token` fournit la structure de lexèmes abstraits utilisée par l'analyseur syntaxique.
- La classe `Lexer` (A CORRIGER section 6) fournit l'analyseur lexical utilisé par les analyseurs syntaxiques.
- La classe `InfixParser` (A COMPLETER section 7) fournit le squelette d'analyseur syntaxique en notation infixe.